This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2024.3408905

Multi-Dimensional Flat Indexing for Encrypted Data

Sabrina De Capitani di Vimercati, *Senior Member, IEEE,* Dario Facchinetti, *Member, IEEE,* Sara Foresti, *Senior Member, IEEE,* Gianluca Oldani, *Member, IEEE,* Stefano Paraboschi, *Member, IEEE,* Matthew Rossi, *Member, IEEE,* Pierangela Samarati, *Fellow, IEEE*

Abstract—We address the problem of indexing encrypted data outsourced to an external cloud server to support server-side execution of multi-attribute queries. Our approach partitions the dataset in groups with the same number of tuples, and associates all tuples in a group with the same combination of index values, so to guarantee protection against static inferences. Our indexing approach does not require any modifications to the server-side software stack, and requires limited storage at the client for query support. The experimental evaluation considers, for the storage of the encrypted and indexed dataset, both a relational database (PostgreSQL) and a key-value database (Redis). We carried out extensive experiments evaluating client-storage requirements and query performance. The experimental results confirm the efficiency of our solution. The proposal is supported by an open source implementation.

Index Terms—Data outsourcing, multi-dimensional index, encrypted data, efficient query execution

1 Introduction

The use of external cloud providers for storing and managing databases is no more an emerging direction, but a widely adopted solution for many individual and business scenarios. Since information in the database to be outsourced can be sensitive, proprietary, or company-confidential, encryption is typically used to protect data confidentiality against the cloud server storing and managing the data.

In this so called, *honest-but-curious* scenario, data remain unintelligible from the cloud server that can only operate on their encrypted representation. However, encryption affects the functionality and the efficient support for fine-grained access and retrieval of data. This problem has been under investigation for more than twenty years and many interesting directions have been investigated, including searchable encryption (e.g., [1]), trusted hardware (e.g., [2], [3]), and coded metadata working as indexes for the evaluation of conditions (e.g., [4], [5]). All these approaches represent valid alternatives depending on the application scenario, but each bears open problems and challenges, and the level of their application in practice is still below the expectations of the research community. Important obstacles to their wide adoption are the performance impact, the limited integration with classical database technology and limited support for query functionality. At the same time, a new push to the development of practical solutions for effectively supporting queries over encrypted data is represented by recent significant technological advancements, including the wide availability of high-bandwidth inexpensive network connections, novel efficient data management solutions for server-side storage, and the increase in the memory and computational capacity available on clients. These advancements introduce novel opportunities for the design of indexing structures for supporting query execution on encrypted data, offering flexibility and performance. In this paper, we leverage such technological advancements to design an indexing structure for effective and efficient execution of queries over encrypted data, which does not require modifications to the server-side software stack and is independent from the nature of server-side storage platforms (i.e., relational or key-value).

Indexes over encrypted data provide a coding for the attributes, so to enable evaluation of conditions on them while not exposing actual values to the storing server. Indexing must however be done carefully to ensure it does not leak information. For instance, while coding protects actual values, a one-to-one correspondence between plaintext values and indexes clearly makes indexes exposed to frequency-based attacks (exploiting profile of occurrences of values or their combination, which would be indeed maintained in a one-to-one indexing). Also, an order-preserving index to support range queries would maintain the order of values in the indexing, hence again leaking information that can enable reconstructing the values behind the indexes. Hence, indexes should not leak, in their values, any order.

Frequency-based attacks can be counteracted by destroying the frequency-based correlation between values and indexes. The extreme case for this is a one-to-many correspondence (i.e., mapping different occurrences of the same value to multiple indexes) with no index value appearing more than once. While destroying frequencies in the index values, such an approach would clearly prove to be cumbersome in query execution. Confusion of frequencies obtained through collision with a many-to-one correspondence (mapping different plaintext values to the same index) is by itself not sufficient since high-occurring values would remain exposed.

- *S. De Capitani di Vimercati, S. Foresti, and P. Samarati are with the Università degli Studi di Milano, Italy.*
  E-mail: firstname.lastname@*unimi.it*
- *D. Facchinetti, G. Oldani, S. Paraboschi, and M. Rossi are with the Università degli Studi di Bergamo, Italy.*
  E-mail: firstname.lastname@*unibg.it*

An effective solution to the problem of protecting against frequency-based attacks is to provide indexing while ensuring a completely flat occurrence of index values through both multiple index values for the same plaintext value as well as collision, that is, through a many-to-many correspondence between plaintext values and indexes with flat index occurrences so to provide confusion and indistinguishability. Importantly, to provide effective protection, not only individual attributes, but also any combination of them, should enjoy a flat frequency distribution Unfortunately, the design of such privacy-preserving indexes over encrypted data is far from being trivial and entails several interrelated challenges. First, as noted, not only individual attributes, but also any combination of them, should be designed to ensure protection against inferences, hence introducing an inevitable curse of dimensionality. Second, there is the need to guarantee effectiveness of indexes (in terms of queries supported and limited overhead caused by spurious tuples returned to the client due to index collisions) and efficiency (in terms of low performance overhead) for query execution. Third, there is the need to limit the storage required at the client for the indexes supporting query evaluation.

In this paper, we address the challenges above and present a novel approach for *multi-dimensional indexing* that: is robust against static inference exposure, performs well in query execution (with support for point and range queries even involving multiple attributes), and requires limited storage at the client side.

The remainder of the paper is organized as follows. Section 2 describes the considered scenario and the rationale of our approach. Section 3 illustrates our approach to cluster tuples for indexing based on flat horizontal partitioning of the original relation. Section 4 presents the definition of indexes and of the data to be stored at the server and at the client to enable query evaluation. Section 5 illustrates the implementation and the extensive experimental evaluation, confirming the effectiveness and applicability of our approach. Section 6 discusses related work. Finally, Section 7 concludes the paper. Appendixes A and B, available as supplementary material, present the procedure used to guarantee a valid flat partitioning of the original relation, and the proofs of theorems, respectively. The artifact of the software and the scripts that permit the reproduction of all the experiments reported in the paper are available open-source at https://github.com/unibg-seclab/flat-index.

2 Scenario and Rationale of the Approach

We frame our work in the context of relational database systems, the most common and well-known technology for the management of large data collections, and illustrate our approach with reference to the outsourcing to the cloud of a relation r over schema $R(a_1, \ldots, a_m)$, with a_j an attribute of r, $j = 1, \ldots, m$. Our problem is the definition of privacy-preserving indexes for enabling execution of queries involving evaluation of conditions over attributes, considering point (i.e., $=$) as well as range (i.e., $>, \geq, <, \leq$) conditions. As running example, we consider the problem of outsourcing the relation in Figure 1(a), where queries may need to evaluate conditions over attributes `State` (the domain is the set of the two-letter codes for states in the USA), and `Age`. A query we want to support is, for example, "SELECT `Name`, `Age` FROM r WHERE `State`="Ca" AND `Age`>38". Figure 1(b) shows a flat indexing with collisions for the relation in Figure 1(a), where index values are represented with Greek letters. Figure 1(c) shows the encrypted and indexed version of the relation in Figure 1(a) to be outsourced at the cloud server, where the encrypted groups of tuples are represented with a gray background.

	Name	Age	State
t_1	Abe	34	Ne
t_2	Bud	34	Tx
t_3	Coy	40	Ne
t_4	Doc	34	Wy
t_5	Edd	37	Ca
t_6	Fox	40	Ak
t_7	Gus	43	Ca
t_8	Hae	46	Ca
t_9	Isa	49	Oh
t_{10}	Jim	55	Wy
t_{11}	Ken	46	Mi
t_{12}	Luc	52	Tx

(a)

I_{Age}	I_{State}	Tuple
ϵ	α	t_1
ϵ	α	t_2
ϵ	α	t_3
ζ	β	t_4
ζ	β	t_5
ζ	β	t_6
η	λ	t_7
η	λ	t_8
η	λ	t_9
θ	δ	t_{10}
θ	δ	t_{11}
θ	δ	t_{12}

(b)

I_{Age}	I_{State}	Enc block
ϵ	α	$t_1t_2t_3$
ζ	β	$t_4t_5t_6$
η	λ	$t_7t_8t_9$
θ	δ	$t_{10}t_{11}t_{12}$

(c)

Fig. 1: Plaintext relation (a), corresponding encrypted and indexed version (b), and relation stored at the server (c)

The goal of our approach is therefore to define an indexing with collision that both: *i)* enjoys flat frequencies of occurrences of index values and combinations thereof and *ii)* performs well for query execution, providing support for both point and range queries, also when multiple attributes are involved. Such protection and efficiency are achieved by the careful grouping of tuples for index definition which employs a recursive multi-dimensional process. Collision and flattening of indexes produces groups of tuples that remain indistinguishable one from the other. Such indistinguishability is maintained at the physical level by operating encryption at the level of groups of tuples through a semantically secure encryption and the application of padding to produce groups all of identical size. The efficiency of the approach is maintained and favored by a careful realization of the index at the server.

Our approach tackles the different challenges involved in the definition and construction of indexing as well as its realization, addressing the involved challenges in different steps.

- *Partitioning.* The first step of our approach is to partition tuples for indexing. Aiming at a flat indexing, the challenge is providing a partitioning suitable for query execution (tuples in the same group will be mapped to the same combination of index values) and that enjoys flat cardinality of groups, that is, all groups have the same number of tuples (with the difference of at most one tuple). The cardinality of the groups is a parameter (k) of the partitioning process that can be arbitrarily set by the data owner (intuitively, it corresponds to a privacy degree provided by the fact that more tuples collide in a same group). Our approach to partition tuples for indexing, ensuring effective and efficient query execution, leverages a spatial representation of the

tuples in a multi-dimensional space. Partitioning is enforced recursively, operating at each a step a cut along one dimension of the multi-dimensional space, and proceeding recursively on each of the subspaces so produced until all subspaces contain k (or $k+1$) tuples.

- *Index construction.* With the partitioning producing groups taking into account multi-attribute values so to accommodate query execution, the next challenge is the realization of the indexing taking into account the overhead in terms of storage and computation at the client side for maintaining indexing information and for processing queries. Our solution for the definition of indexes to be associated with groups comprises two alternatives: value-based and group-based indexing, both enjoying limited overhead and each potentially to be preferred over the other one depending on architectural considerations. For both solutions, at the client side, only a compact map needs to be maintained for translating queries on plaintext data into queries operating on indexes stored at the server.
- *Seamless realization.* The third challenge we address is the seamless realization of the approach over current architectural solutions, to enable the use of privacy-preserving indexing over existing storage and computational cloud services. We support and evaluate both relational (PostgreSQL) as well as key-value (Redis) data management technologies. For both, we illustrate the organization of the storage and the execution of queries. Our extensive experimental evaluation demonstrates the effectiveness and the efficiency of our approach.

The remainder of the paper is organized following the steps above, illustrating partitioning (Section 3), index construction and their seamless realization (Section 4), and our implementation and experimental evaluation (Section 5).

3 Partitioning

Our approach for the construction of index values is based on a partitioning of the tuples in the original relation into groups of a fixed number of tuples. All the tuples in the same group are then associated with the same combination of index values. The number of tuples that must be included in each group, denoted k, is a parameter that can be arbitrarily set. Clearly, a larger k provides more protection, but also increases the potential overhead of query execution (we will elaborate more on this in Section 5). In the following, we introduce the concept of k-flat partitioning (Section 3.1), illustrate how to recursively partition a relation (Section 3.2), and present our approach for the computation of a k-flat partitioning (Section 3.3).

3.1 k-flat partition

The first step of our approach is the partitioning of tuples in groups of the same size k. Since the cardinality of the relation may not be a multiple of k, we need to account for the remainders, which we accommodate by allowing groups to include at most one tuple more than the k requested (as needed to fully cover the set of tuples to be partitioned). Our definition of k-flat partition captures the partitioning of tuples to produce a *maximal* flattening of groups with cardinality k as follows.

Definition 3.1 (k-flat partition) Let r be a relation, and k be a natural number. A *k-flat partition* of r, denoted $\mathcal{P}$, is a partition $\mathcal{P} = \{g_1, \ldots, g_p\}$ of tuples in r such that:

1) $\forall g \in \mathcal{P}, k \leq \mathrm{card}(g) \leq k+1$;
2) $p = \lfloor \mathrm{card}(r)/k \rfloor$.

The first condition expresses the requirement on the cardinality of the groups (allowing groups to have either k or $k+1$ tuples, this latter being needed to accommodate remainders), and the second condition dictates the number of groups to be the *maximum* among those that satisfy condition 1, or - equivalently - the number of groups with $k+1$ tuples to be *minimum*. By dictating the number of groups in the partition, the second condition forces exactly h=(card(r) mod k) of the groups to have $k+1$ tuples, while all the others will have k tuples. In other words, the condition rules out from consideration partitions that do not enjoy maximum flattening, that is, that have a number of groups of cardinality $k+1$ larger than the number of remainders to be accommodated. For instance, assume $\mathrm{card}(r)$=231 and k=10. Condition 2 would accept only a partition composed of 23 groups (one of which composed of 11 tuples, all the others being of 10 tuples) ruling out of consideration partitions composed of 22 groups (eleven of which composed of 11 tuples) or 21 groups (all with 11 tuples), which - although satisfying condition 1 - do not maximize the required flattening of $k = 10$.

Clearly, for a relation r to have a k-flat partition, the number of remainders to be accommodated (i.e., the extra tuples to allocate to groups) must be not greater than the number of groups composing the partition. For instance, trivially, no k-flat partition for $k = 10$ can exist for a relation with 23 tuples. In other words, with h=(card(r) mod k) and $p = \lfloor \mathrm{card}(r)/k \rfloor$, it must be that $h \leq p$, which is also a sufficient condition for a k-flat partition to exist, as stated by the following theorem.

Theorem 1 (Existence of a k-flat partition) Let r be a relation and k be a natural number such that $\mathrm{card}(r) \geq k$. A k-flat partition $\mathcal{P}$ of r exists iff $h \leq p$, with h=($\mathrm{card}(r)$ mod k) and p=$\lfloor \mathrm{card}(r)/k \rfloor$.

Given a relation r and a natural number k, we say that r is *k-valid* if a k-flat partition exists for r. This is captured by the following definition.

Definition 3.2 (Validity) Let r be a relation, and k be a natural number. Relation r is said to be *k-valid* iff $h \leq p$, with h=($\mathrm{card}(r)$ mod k) and p=$\lfloor \mathrm{card}(r)/k \rfloor$.

While the observation in Theorem 1 may seem a non-issue for the computation of a k-flat partition of r since the cardinality of r is extremely large and k is very small, it is an important aspect to take into account in the partitioning process, which, if not done properly, may easily degenerate.

Our approach to compute a k-flat partition is via a process recursively cutting a relation in two groups at each step, until a k-flat partition is reached. In the following,

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2024.3408905

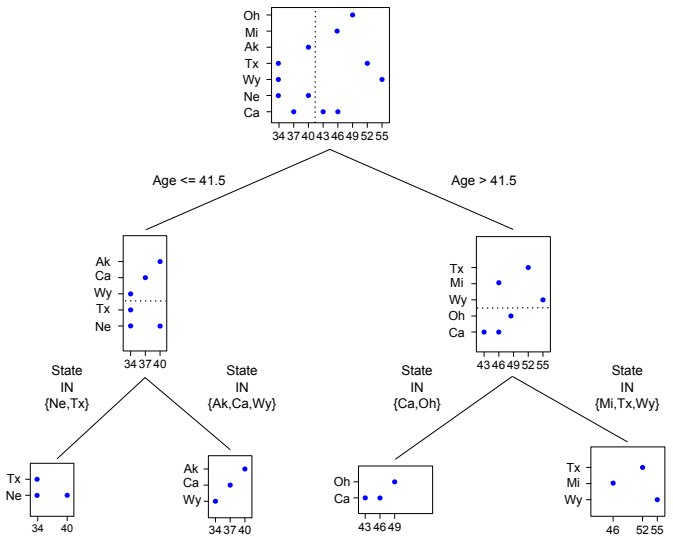


Fig. 2: Graphical representation of the cuts performed by procedure **Cut** over the relation in Figure 1(a)

when clear from the context, we will use the terms relation and group interchangeably.

To ensure that our recursive process terminates with the computation of a k-flat partition, we force the cut at each step to produce only k-valid relations and to not increase the number of groups with cardinality $k+1$. We then introduce the notion of cut validity as follows.

Definition 3.3 (Cut validity) Let r be a relation and k be a natural number. A cut (r_l,r_r), partitioning r into two groups, is *valid* iff both r_l and r_r are k-valid (Definition 3.2) and $h = h_l + h_r$, with h=($\mathrm{card}(r)$ mod k), h_l=($\mathrm{card}(r_l)$ mod k), and h_r=($\mathrm{card}(r_r)$ mod k)

Intuitively, a cut is valid if the two relations resulting from it are k-valid, that is, a k-flat partition exists for them, and the total number of groups of cardinality $k+1$ is not increased by the cut. For instance, consider a relation composed of 233 tuples, and k=10. A cut partitioning it into two relations of 23 and 210 tuples, respectively, is not valid due to the non validity of the first relation (which cannot have a 10-flat partition). Also, a cut partitioning it into two relations of 117 and 116 tuples, respectively, is not valid since their k-flat partitions, having respectively seven and six groups of 11 tuples, cannot represent a k-flat partition of the original relation. We note that each relation r with more than k tuples has at least a valid cut, as stated by the following theorem.

Theorem 2 (Valid cut existence) Let r be a k-valid relation with $\mathrm{card}(r) \geq k$. There always exists a valid cut for r.

Also, any k-flat partition of the relations resulting from a valid cut of r represents a k-flat partition for r, as stated by the following theorem.

Theorem 3 (k-flat composition) Let r be a k-valid relation for a natural number k, (r_l,r_r) a valid cut for it, $\mathcal{P}_{r_l}$ a k-flat partition of r_l, and $\mathcal{P}_{r_r}$ a k-flat partition of r_r. $\mathcal{P}$=$\mathcal{P}_{r_l} \cup \mathcal{P}_{r_r}$ is a k-flat partition for r.

Since our problem is to group tuples for index construction, it is important not only to partition tuples as a k-flat partition to ensure flat indexing, but also to group them in a way that performs well with respect to query execution. Intuitively, a partitioning maintaining tuples with the same or close values for an attribute within the same group as much as possible behaves better, meaning it introduces less performance overhead in the execution of queries involving that attribute, than an approach scattering such values in different groups. However, as already noted, with multiple attributes involved, the problem is far from being trivial, as each dimension represents a candidate to consider.

We introduce our approach by first describing how we take into consideration the values within tuples so to provide a partitioning performing well for query execution, and then describing its tweaking to enforce partitioning to ensure k-flatness.

3.2 Recursive partitioning

Our approach to partition leverages a representation of the dataset in a multi-dimensional space and enforces partitioning through recursive cuts, similarly to what is done in multi-dimensional anonymization approaches (e.g., Mondrian [6]) and in some multi-dimensional indexing approaches, like quad trees, k-d trees and R-trees. Our problem and solution bears however several important differences. As a matter of fact, we need to cluster tuples to produce indexing performing well for query execution (in contrast to cluster tuples for data generalization), while ensuring groups with flat occurrences (in contrast to just require a minimum group cardinality). Our approach performs recursive cuts considering then a flexible and dynamic order of values in the different dimensions, and also enforcing controls and adjustments to ensure flat partitioning as per Definition 3.1.

Our partitioning process works then in a multi-dimensional space, with one dimension for each attribute to be indexed, and where each tuple is the point in such a space where its coordinate values (i.e., the values of its attributes) meet. As an example, the space appearing at the top of Figure 2 is the multi-dimensional representation of attributes `State` and `Age` for the tuples in Figure 1(a). For the attributes to be indexed, a point in the multi-dimensional space can correspond to more tuples, which can be represented as a counter associated with the point. Since in our example such a value is always 1, we simply omit it. Note that the tree in Figure 2 is just a representation of the recursive calls of the cutting process and of the subspaces it produces, and does not represent the indexing structure itself, which is defined in a subsequent step over the subspaces in the leaves.

For the partitioning process and index construction, we classify attributes to be indexed into two categories:

- *continuous* attributes (e.g., `Age` in Figure 1(a)), characterized by a total order relationship on their domain, and on which range conditions need to be supported;
- *nominal* attributes (e.g., `State` in Figure 1(a)), which do not have an order in their domain and hence on which only equality conditions apply. The domain

can support queries for a set of values, all explicitly represented in the condition.

While the spatial representation conveys an order of values along a dimension, we maintain such an order fixed, and corresponding to the order dictated by the domain, only for continuous attributes, so that partitioning will cluster together same or close values. By contrast, we adjust the order of nominal attributes as best suited for the process, as we elaborate next.

The partitioning process works by cutting at each step the tuples along one dimension (attribute) in the space and recursively calling itself on each of the two produced subspaces. At each iteration, the dimension along which a cut is to be performed is chosen to be an attribute that enjoys the highest number of distinct values. If the attribute is a continuous attribute, the cut divides the tuples into two groups depending on their value with respect to the median: tuples with values lower than or equal to the median in one group and tuples with values higher than the median in the other group. Should the median correspond to the maximum value for the attribute in the relation, the values equal to the median will be put into the second group (which would otherwise be empty) instead of the first one. If the attribute is a nominal attribute, the cut divides the tuples into two groups with a "first-fit decreasing" bin packing strategy [7], considering values of the attribute in decreasing order of their frequencies and placing tuples that have the value under consideration in the smaller group. Figure 2 illustrates the working of the partitioning process for our running example aiming at a 3-flat partition. The first cut operates on attribute `Age` (which has 8 distinct values), splitting tuples in two groups, the left group has the tuples with `Age` lower than or equal to the median (which is 41.5) and the right group has the tuples with `Age` higher than the median. On each of the two spaces, the subsequent cut operates on attribute `State`, dividing tuples into two groups, considering `State` values in decreasing order of occurrences and, for each `State` value under consideration, placing tuples with such value in the group that is smaller. In the figure, the order of values in the `State` dimension has been rearranged at each step to better represent the cut graphically (starting from the origin, they always appear in decreasing order of occurrences). The resulting groups, reported at the bottom of Figure 2, have all cardinality 3, and hence no further cut needs to be performed.

3.3 Computing a k-flat partition

Our approach to compute a k-flat partition of a relation r uses recursive partitioning as illustrated above, enriched to ensure the validity of the cut performed at each step and the enforcement of possible adjustments if the cut is not valid. Figure 3 illustrates the pseudocode of the process, which comprises three procedures: **Partition**, **Cut**, and **Check**.

Partition. It performs the partitioning recursively calling itself and calling procedure **Cut** for performing the cutting process described above, eventually determining a k-flat partition $\mathcal{P}$. When called, **Partition**(r) first evaluates the cardinality of r (line 1). If such a cardinality is not greater than $k+1$ (i.e., it is either k or $k+1$), no further cut needs

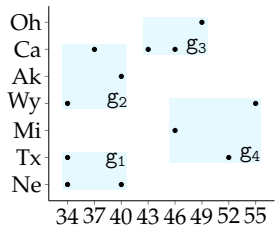
INPUT: (r, A, k) /* relation r to partition; attributes A to index; global var. k */
OUTPUT: $\mathcal{P}$ /* k-flat partition $\mathcal{P}$ of r */
PARTITION(r)
1: **if** $\text{card}(r) \leq k+1$ **then** $\mathcal{P} := \mathcal{P} \cup \{r\}$
2: **elseif** count(distinct A) = 1 **then** /* all tuples over A are equal */
3: $p := \lfloor \text{card}(r)/k \rfloor$
4: $h := \text{card}(r) \bmod k$
5: Let $\{g_1, \ldots, g_p\}$ be a partition of r in h groups of $k+1$ tuples
6: and $p-h$ groups of k tuples
7: $\mathcal{P} := \mathcal{P} \cup \{g_1\} \cup \ldots \cup \{g_p\}$
8: **else**
9: Choose $a \in A$ s.t. count(distinct a) is maximum
10: (r_l, r_r) := **Cut**(r,a)
11: **Partition**(r_l)
12: **Partition**(r_r)

CUT(r,a) /* cut relation r over attribute a in two k-valid relations r_l, r_r */
1: **if** a is continuous **then**
2: *med* := median($r[a]$) /* compute the median of a */
3: **if** *med* = max($r[a]$) **then**
4: $r_l := \{t \in r \mid t[a] < med\}$; $r_r := \{t \in r \mid t[a] \geq med\}$
5: **else** $r_l := \{t \in r \mid t[a] \leq med\}$; $r_r := \{t \in r \mid t[a] > med\}$
6: m := **Check**(r,r_l,r_r)
7: **case** m **of** /* move m tuples to produce two k-valid relations r_l, r_r */
8: > 0: Move m tuples with values for a closest to *med* from r_l to r_r
9: < 0: Move m tuples with values for a closest to *med* from r_r to r_l
10: **else** /* a is nominal */
11: $\forall v \in r[a]$, c_v := count($r[a]$=v) /* count c_v to be priority of v */
12: Let Q be a max priority queue with the distinct values in $r[a]$
13: $r_l := \emptyset$; $r_r := \emptyset$
14: **while** NOTEMPTY(Q)
15: v := POP(Q)
16: **if** $\text{card}(r_l) < \text{card}(r_r)$ **then** $r_l := r_l \cup \{t \in r \mid t[a] = v\}$
17: **else** $r_r := r_r \cup \{t \in r \mid t[a] = v\}$
18: m := **Check**(r,r_l,r_r)
19: **case** m **of** /* move m tuples to produce two k-valid relations r_l, r_r */
20: > 0: Move m tuples with the minimum count from r_l to r_r
21: < 0: Move m tuples with the minimum count from r_r to r_l
22: **return** r_l,r_r

Fig. 3: Algorithm for computing a k-flat partition

to be performed and r is added to $\mathcal{P}$. Else, if all the tuples in r have the same values for all the attributes in the set A of attributes to index (line 2), it simply splits the tuples in $\lfloor \text{card}(r)/k \rfloor$ groups each containing either k or $k+1$ tuples (as per Definition 3.1). Otherwise (line 8) it picks an attribute a with the highest number of distinct values and calls procedure **Cut** to split the tuples in the relation along a's dimension, then recursively calling itself on the two returned groups.

Cut. Called with a relation r and attribute a as parameters, procedure **Cut** partitions the tuples in r based on the values of a, enforcing the process described in Section 3.2, distinguishing the cases where a is continuous (lines 1-9) or nominal (lines 10-21). After producing the two groups r_l and r_r, it calls procedure **Check** (lines 6 and 18), which checks the validity of the computed cut and returns the number m of tuples to be moved from a group to the other to make the cut valid (in case it is not), while minimizing the number of tuples to be moved. The sign ($+$ or $-$) of the returned number indicates the direction of the movement: a positive number indicates that tuples need to be moved from r_l to r_r, while a negative number indicates that tuples need to be moved from r_r to r_l (while 0 is returned if the cut is already valid). To maintain the quality of the computed cut, the m tuples to be moved from one group to the other are those close to the median if the cut was on a continuous attribute (lines 7-9), or those with a value v for a with a lower number of occurrences if the cut was on a nominal attribute (lines 19-21).



(a) Spatial representation

MAP

Gid	Age	State	Tuples
g_1	[34,40]	NeTx	$t_1t_2t_3$
g_2	[34,40]	AkCaWy	$t_4t_5t_6$
g_3	[43,49]	CaOh	$t_7t_8t_9$
g_4	[46,55]	MiTxWy	$t_{10}t_{11}t_{12}$

(b) MAP

I_{Age}	I_{State}	Tuples
ϵ	α	$t_1t_2t_3$
ζ	β	$t_4t_5t_6$
η	λ	$t_7t_8t_9$
θ	δ	$t_{10}t_{11}t_{12}$

(c) Value-based

I_G	Tuples
γ_1	$t_1t_2t_3$
γ_2	$t_4t_5t_6$
γ_3	$t_7t_8t_9$
γ_4	$t_{10}t_{11}t_{12}$

(d) Group-based

Fig. 4: Spatial representation of the coverages of the running example (a), corresponding MAP (b), and its value-based (c) and group-based (d) indexing

Value-based

SALT

Att	Salt
Age	σ_{Age}
State	σ_{State}

INDEX_Age

Coverage	Freq	Token
[34,40]	2	$\tau_{[34,40]}$
[43,49]	1	$\tau_{[43,49]}$
[46,55]	1	$\tau_{[46,55]}$

INDEX_State

Coverage	Freq	Token
AkCaWy	1	τ_{AkCaWy}
CaOh	1	τ_{CaOh}
MiTxWy	1	τ_{MiTxWy}
NeTx	1	τ_{NeTx}

Group-based

INDEX_Age

Coverage	Gid
[34,40]	g_1, g_2
[43,49]	g_3
[46,55]	g_4

INDEX_State

Coverage	Gid
AkCaWy	g_2
CaOh	g_3
MiTxWy	g_4
NeTx	g_1

Fig. 5: Information stored at the client

The pseudocode of procedure **Check** and a detailed description of its working, distinguishing the different cases of non-validity for a cut and hence of minimum number of tuples to be moved from one group to the other (and viceversa) to make it valid, are illustrated in Appendix A.

Theorem 4 (k-flat partition computation correctness) Let r be a k-valid relation for a natural number k. **Partition**(r) terminates and computes a k-flat partition for r.

4 INDEXING AND ENCRYPTION

At the end of the partitioning process, each group in the k-flat partition contains tuples that must be mapped to the same combination of index values. The next step is then the definition of such indexes (Section 4.1), the construction of the data structures to be maintained at the client for supporting query evaluation (Section 4.2), and the organization of the encrypted and indexed data to be stored at the cloud server (Section 4.3).

4.1 Map construction

We start by identifying, for each attribute and each group of tuples in the partition, the attribute values that the group covers, specified as an interval for a continuous attribute and as a set of values for a nominal attribute.

Definition 4.1 (Coverage) Let $\mathcal{P}$ be a k-flat partition of a relation r, $a \in A$ be an attribute to index, and g be a group in $\mathcal{P}$. The *coverage* of a in g, denoted $g[a]$, is defined as:

- $g[a]=[v_l,v_u]$, with $v_l=min\{t[a] \mid t \in g\}$ and $v_u=max\{t[a] \mid t \in g\}$, if a is a continuous attribute;
- $g[a]=\{t[a] \mid t \in g\}$, if a is a nominal attribute.

For instance, with reference to the partitioning process in Figure 2, whose result is graphically illustrated by the spatial representation in Figure 4(a), g_1[Age]=[34,40], g_1[State]={Ne,Tx}. We refer to the groups in a k-flat partition, together with their coverages for the attributes to index and the tuples in each group, as the MAP of the partition, formally defined as follows.

Definition 4.2 (Map) Let $\mathcal{P}$ be a k-flat partition of a relation r and $A = \{a_1, \ldots, a_n\}$ be a set of attributes to index. The MAP of $\mathcal{P}$ over A is the set of tuples $\{\langle g[gid], g[a_1], \ldots, g[a_n], g[tuples]\rangle \mid g \in \mathcal{P}\}$, with $g[gid]$ the unique group identifier of g, and $g[tuples]$ the set of tuples in g.

Figure 4(b) reports the MAP for the partition in Figure 4(a). For simplicity, in the figure and in the remainder of the paper, we omit the brackets and commas in the coverage of nominal attributes. For instance, NeTx stands for {Ne,Tx}.

In the following, we use notation MAP[gid] to denote the set of all gid of the groups in $\mathcal{P}$, MAP[a] to denote the support of multiset $\bigcup_{i=1}^{p} g_i[a]$, and $\mu_a(c)$ to denote the multiplicity of coverage c of a in the multiset. For instance, with reference to Figure 4(b), MAP[gid]={g_1, g_2, g_3, g_4}, MAP[Age]={[34,40], [43,49], [46,55]}, with $\mu_{Age}([34,40])$=2 and $\mu_{Age}([43,49])=\mu_{Age}([46,55])$=1.

To define indexes at the level of group of tuples (all tuples in a group are to be associated with the same combination of indexes), we define indexes over the MAP. We investigate two approaches to indexing: value-based (indexing coverages) and group-based (indexing group ids), which we then evaluate with respect to the size of the storage required for the client and the performance in query evaluation (Section 5).

With value-based indexing, indexes are computed with respect to coverages (hence producing the same combination of index values for the tuples in each group), while mapping different occurrences of the same coverage to different index values.

Definition 4.3 (Value-based indexing) Let MAP be a map of a k-flat partition $\mathcal{P}$ of relation r over a set A of attributes to index. A *value-based indexing* over MAP is a set of functions, one for each attribute a in A, defined as ι_a:MAP[a]$\rightarrow 2^{\mathcal{I}_a}$, with $\mathcal{I}_a$ the domain for a of index values, such that:

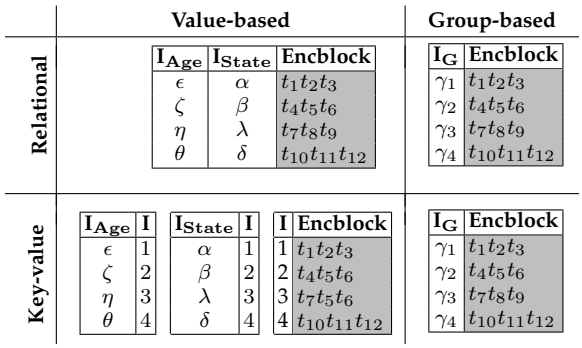


Fig. 6: Information stored at the server

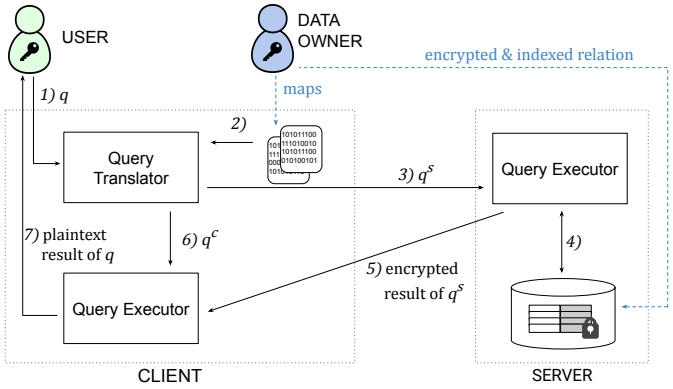


Fig. 7: Query execution process

1) $\forall c \in \text{MAP}[a], |\iota_a(c)| = \mu_a(c)$;
2) $\forall c, c' \in \text{MAP}[a]$, with $c \neq c'$, $\iota_a(c) \cap \iota_a(c') = \emptyset$;
3) $\forall a' \in A$, with $a \neq a'$, $\mathcal{I}_a \cap \mathcal{I}_{a'} = \emptyset$.

In other words, there is a function for each attribute to index, mapping coverages to sets of indexes such that: *1)* each coverage is mapped to as many indexes as the multiplicity of the coverage; *2)* the sets of indexes of different coverages are disjoint, and *3)* the sets of indexes of different attributes are disjoint. Figure 4(c) illustrates an example of value-based indexing for the MAP in Figure 4(b). At the practical level, value-based indexing for an attribute a can be realized by using a salt σ_a for the attribute and a random token τ_c for each of its coverages c, and encrypting (with CBC mode) the token using the salt as initialization vector. Index values are extracted from the result of encryption as fixed-length non-overlapping strings of bits.

With group-based indexing, indexes are computed with respect to group identifiers (hence producing the same index value for the tuples in each group) while mapping different group identifiers to different index values.

Definition 4.4 (Group-based indexing) Let MAP be a map of a k-flat partition $\mathcal{P}$ of relation r over a set A of attributes. A *group-based indexing* over MAP is an injective function ι_{gid}:MAP$[gid] \rightarrow \mathcal{I}_{gid}$, with $\mathcal{I}_{gid}$ the domain of index values.

Figure 4(d) illustrates an example of group-based indexing for the MAP in Figure 4(b). At the practical level, group-based indexing can be realized by simply assigning a sequential number to each group and then applying a random shuffling on all the values; groups are then uploaded to the cloud server in the order of the group identifier. This solution guarantees absence of collisions and the most compact representation of the group identifiers.

4.2 Client-side storage

At the client, a data structure (which we refer to as *client map*) needs to be maintained to enable translation of conditions on plaintext values in the queries into conditions to be executed on the indexed dataset at the server.

For value-based indexing, the client needs to store, for each attribute a to index, the salt σ_a to be used as initialization vector for index generation, and, for each of its coverages c, the multiplicity of the coverage $\mu_a(c)$ (which dictates how many index values the coverage maps to) and the initialization token τ_c. For group-based indexing, the client needs to store, for each attribute, the set of coverages and their corresponding group ids. Figure 5 illustrates the client map for the value-based and group-based indexing of our running example.

Translating plaintext conditions into conditions on index values requires determining the coverages involved in the query evaluation, that is, including plaintext values involved in the query. Depending on the conditions expected to be evaluated, the client map can be organized at the physical level for providing efficient retrieval of such coverages. For instance, coverages for continuous attributes can be stored sorted with respect to their minimum (maximum, resp.) interval value to support efficient evaluation of conditions of the form $a \leq v$ ($a \geq v$, resp.) or as interval trees hence offering a logarithmic cost for searches, at the price however of more storage space (up to three times as much).

For nominal attributes, mapping of plaintext values to coverages can be realized via a bitmap representation of coverages, with a row for each plaintext value in the actual domain of the attribute and a bit for each coverage. Since bitmaps are expected to be sparse, it is advantageous to consider the use of *roaring bitmaps* [8], a recent technique with associated open-source implementation that offers good performance in terms of size and speed for sparse bitmaps. Bitmaps and roaring bitmaps allow the efficient (constant cost) retrieval of all coverages including a value of interest. Since the cost required for the construction of all these alternative structures is low (a few seconds for tables containing millions of tuples) and their size depends on the distribution of data for a given dataset, all the alternative structures can be built and the most compact one chosen.

4.3 Server-side storage

At the server side, the relation to be outsourced can be stored with tuples encrypted and associated with the computed indexes. Since all tuples in a group share the same indexes, tuples within a group are indistinguishable from the indexes, and hence query execution always operates at the granularity of group (either none or all tuples in a group are to be returned). Given this, encryption can be applied at the group level, producing a single encrypted block for the whole group. Thanks to the k-flatness of the partition, encryption at the level of group enjoys a corresponding

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2024.3408905

flatness on the size of encrypted blocks (provided a small padding).

At the physical level, the organization of the encrypted and indexed data depends on the database supported at the server (e.g., relational vs key-value). Figure 6 illustrates the encrypted and indexed representation for the relation in our running example, considering value-based and group-based indexing, assuming the adoption of a relational and of a key-value database.

With a relational database, data can be simply stored as a relation with an attribute for the encrypted block, and an attribute for each index to be supported (see Figure 6).

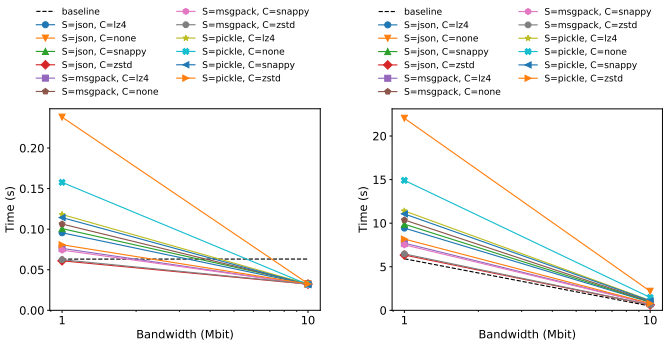
With a key-value database, value-based indexing requires storing different key-value structures: a primary one for the encrypted block and a secondary one for each of the indexed attributes to be supported, all connected via a common id. The common id works as a key for the structure storing the encrypted blocks and as value for the structures reporting the indexes, with each index working as key for the corresponding structure. Group-based indexing is simply realized with a single structure, with as key the index of the group id and as value the encrypted block (see Figure 6). The key-value model turns out to offer a natural mapping for group-based indexing.

Once the encrypted and indexed relation has been stored at the server, each query q formulated at the client side on relation r can be translated into a query q^s working on the outsourced relation. Figure 7 illustrates the query execution process. The translation of q in q^s is performed using the client map. The encrypted tuples retrieved as result of query q^s are sent to the client, decrypted, and filtered through the execution of a query q^c that eliminates possible spurious tuples (i.e., tuples satisfying q^s but not q). Query q^c is the same query as q with the only difference that is executed over the result of q^s and not over relation r.

5 Implementation and Experiments

To verify the effectiveness of our approach, we have realized a prototype and run a series of experiments. In the remainder of this section, we first illustrate the description of the prototype, which supports both a relational (Postgres) and a key-value (Redis) realization of our approach. We then illustrate the experimental results aimed at evaluating the storage required at the client for the client map (Section 5.1) and the impact on performance in query evaluation due to the grouping of tuples (Section 5.2).

Prototype description. Given a dataset to be outsourced, the prototype computes a k-flat partition (Section 3), builds the client map, and generates the encrypted and indexed dataset (Section 4) for its outsourcing at a server supporting either a relational or a key-value database. The prototype is written in Python. The computation of the k-flat partition is realized through a multi-container Docker application leveraging Apache Spark, using Pandas [9] and Arrow [10] for improving its efficiency. The client map is made persistent on disk, serialized using Pickle [11], compressed using the open source Bzip2 library [12], and encrypted using a non-deterministic authenticated encryption cipher. The encryption of each group of tuples in the outsourced relation is obtained by serializing the tuples in JSON format, encoding them in utf-8 and compressing them using the open source Zstandard library [13]. Such combination was chosen after experimenting with various alternatives (JSON, MessagePack and Pickle for serialization and LZ4, Snappy and Zstandard for compression) as it proved to to have the best performance (see Figure 8). The binary object is then padded and encrypted using a non-deterministic authenticated encryption cipher. We rely on Docker Compose [14] to automatically build, install, and run the application. As anticipated at the end of Section 1, all the software is open-source and available on Github.



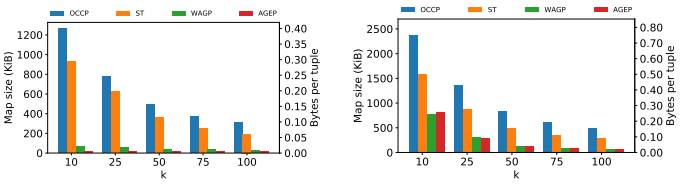
(a) Point queries (b) Range queries

Fig. 8: Performance of serialization and compression alternatives

The client application focuses on the management of queries. The client application rewrites queries expressed over the plaintext relation in queries operating on the encrypted and indexed dataset. The rewritten query is sent to the server application, which is implemented as a separate container. PostgreSQL is used for the relational database implementation and Redis for the key-value implementation. Since Redis does not support key-value stores with composite keys, to implement our value-based approach we relied on the execution of a LUA script on the Redis instance. This script is responsible of retrieving the values of the field connecting all the key-value structures (working as key for the structure storing the encrypted blocks and as value for the structures storing index values) for each attribute in the query. This request is processed in a single interaction, with communication latency equal to a single RTT.

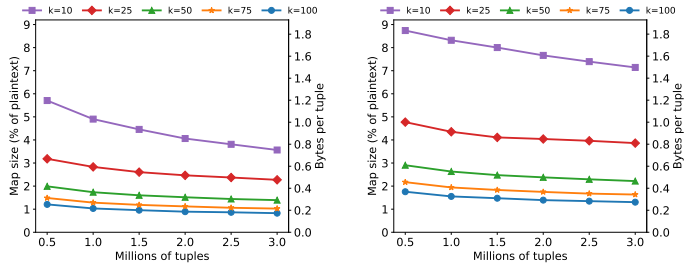
The transmission of the query from the client to the server is implemented using SQLAlchemy [15] or Redispy [16], respectively for PostgreSQL and Redis. A SQLite in-memory database, empty at the start of the application, post processes the result returned by the server to remove spurious tuples (i.e., tuples returned by the server due to index collision but not belonging to the result of the original query).

Dataset and experiment settings. We have performed experiments on the `usa2019` dataset [17], a publicly available large dataset from U.S. Census Bureau comprising more than 3M tuples for a total size of 65MB. For the experiments, we considered a projection of the dataset on nominal attributes `State` (`ST`) and `Occupation` (`OCCP`), and continuous attributes `Age` (`AGEP`) and `Wage` (`WAGP`).



(a) Value-based (b) Group-based

Fig. 9: Size of client maps for each attribute, varying k (left axis: overall; right axis: bytes per tuple)



(a) Value-based (b) Group-based

Fig. 10: Size of client map, varying the size of the dataset (left axis: % of plaintext; right axis: bytes per tuple)

5.1 Client storage

The size of the client-side map is affected by three factors: *1)* the type (i.e., continuous vs nominal), size, and distribution of values of the indexed attributes; *2)* the indexing approach (i.e., value-based vs group-based); and *3)* the value of k. For the first factor, the attributes considered from `usa2019` represent different characteristics. We then run experiments for both value-based and group-based indexing, varying the value of k.

Figure 9 shows the size of the client map for each of the four attributes of the `usa2019` dataset considering the value-based and group-based indexing, varying the value of k. The figure also reports, on the right axis, the average size of the map expressed as bytes per tuple. For both value-based and group-based indexing, the map size for continuous attributes (`WAGP` and `AGEP`) is smaller than the one for nominal attributes (`OCCP` and `ST`). Furthermore, we can observe a significant reduction in the size of the map as k increases, which however implies a higher number of spurious tuples (see next section).

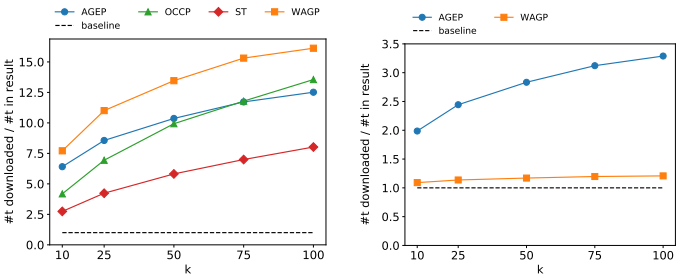
Figure 10 shows the size of the client map (summing the size for all four attributes), in terms of percentage over the size of the dataset (left axis) and of bytes per tuple (right axis) for different values of k, varying the size of the dataset (the percentage and the bytes per tuple for a given database are linearly dependent). The datasets of various size have been obtained extracting random samples from `usa2019`. The graphs show that the size of the map decreases at the increase of k. As visible from the graphs, maps created for larger datasets (while being larger in absolute size) occupy a smaller percentage of the size of the plaintext dataset. This is maintained over all the samples, and has been confirmed from other experiments we ran on different datasets as representative of small and very large datasets, namely `usa2018` [18] (0.5M tuples of 12 MB) and `transactions` [19] (sample of 30M tuples of 1.5 GB). For instance, for k=25, the size of the map in terms of percentage over the size of the dataset, reporting in the order `usa2018`, `usa2019`, and `transactions` is: 4.51%, 2.10%, and 1.40% (for value-based indexing), and 6.66%, 3.60%, and 2.10% (for group-based indexing). Note that, as size of the dataset, we considered the projection over the indexed attributes, while the actual size of the dataset is much larger (containing also all not indexed attributes). The size of the client map compared with the size of the dataset is then in practice even much smaller than what observed in our experiments. As it can be observed from the reported numbers, and as also visible from Figure 10, group-based indexing requires, in the examined datasets, between 50% and 100% more client-side storage than value-based indexing. However, as we will see in the next section, it consistently offers better performance.

5.2 Performance

The indexes constructed as illustrated in the previous sections trivially guarantee that all tuples responding to the original queries are returned in the encrypted result retrieved from the server. However, by design, index collision (i.e., the fact that different values are mapped to a same index), clearly implies retrieval of additional tuples that do not belong to the result of the original queries. These are removed by the client by re-applying the query locally as a post processing step [5], [20]. Such additional tuples bring a potential overhead in query execution due for communication and processing. We discuss first the evaluation with respect to the number of additional tuples and then the execution time, comparing them with respect to the realization of the queries on plaintext values (i.e., offering no protection on the database content). We also discuss the impact of latency and bandwidth.

For evaluating performance, we run different sets of experiments. Each experiment executes in sequence a sample of queries randomly extracted from a pool of 5.000 queries. The queries in the pool are grouped according to their selectivity. Our experiments consider queries with a selectivity of up to 10% of the dataset. These are the most interesting configurations, where indexes are useful to filter tuples in query results; queries that return a larger portion of the dataset may lead to a flat retrieval of the whole dataset.

Additional tuples. We have first evaluated the overhead, in terms of additional number of tuples downloaded from the server, for point queries (for all the four attributes of the `usa2019` dataset) as well as for range queries (for the two continuous attributes). Figure 11 reports the ratio between the number of tuples in the groups retrieved from the server and the tuples actually belonging to the query result; the horizontal baseline at value 1.0 represents the profile of a query executed on a plaintext database. As visible from Figure 11, the number of spurious tuples increases with the increase of k (the larger the groups the greater the number of tuples returned due to index collision that do not belong to the result). However, we note that its limited value with respect to k (the worst overhead is for `WAGP` reaching 15x over the baseline for k=100), and the limited overhead for



(a) Point queries (b) Range queries

Fig. 11: Overhead in the number of tuples downloaded

range queries thanks to the multi-dimensional space partitioning used for defining groups. Most importantly, as we will show next, the overhead in terms of number of tuples shows a much lower impact in terms of execution time. Current networks offer a relatively high bandwidth and the overhead introduced by spurious tuples is dominated by other factors. In addition, the use of data compression in the storage of tuples significantly reduces the impact of spurious tuples.

Execution time. We have evaluated the performance for both value-based and group-based indexing, building both a PostgreSQL and a Redis implementation for the server, hence considering four different configurations for the realization of our solution. The experiments aimed at comparing performance for the different configurations with respect to the one of the baseline configuration, which corresponds to a plaintext dataset stored in PostgreSQL where queries are executed on the plaintext dataset without any rewriting, and classical indexes are defined within the database over the attributes involved in queries. We do not present a baseline configuration with Redis, because the key-value model does not support queries on attributes other than the key.

We have then evaluated the execution time for point and range *single attribute* queries over continuous attribute `WAGP` and nominal attribute `OCCP`, as well as of *multi-attribute* queries involving both the attributes for various configurations obtained varying k (for multi-attribute queries, we evaluate a conjunction between the selection predicates on the attributes). We have measured both the *server execution time* and the *global execution time*. The global execution time measures the overall time required to: submit the query to the client-side query translator; parse the query; generate the index values and rewrite the query; submit the query to the server; execute the query on the server; send the resulting encrypted blocks to the client; decrypt the encrypted blocks and serialize the resulting plaintext tuples in SQLite; remove spurious tuples. The server-side execution time measures only the time required by the server to run the query and retrieve the encrypted blocks. In the first set of experiments, the network latency between client and server is set to 10 ms (a value that assumes the server to be relatively near to the client), and the bandwidth to 1 Gbps (a value representative of current network connections; we will explore next the impact of latency and bandwidth on performance).

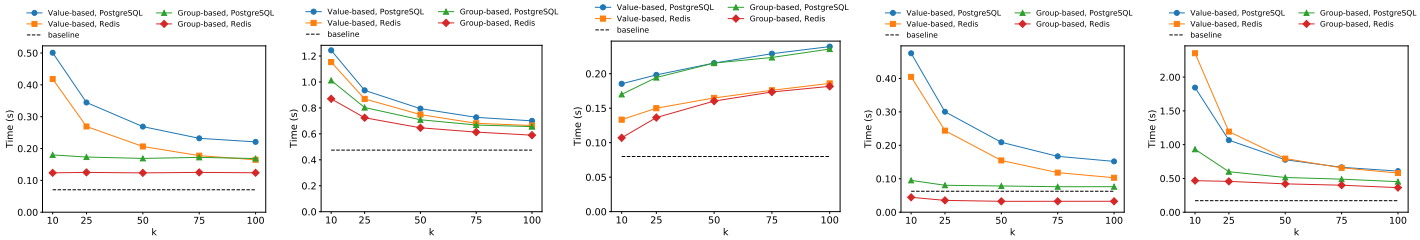
The results reported are the average observed execution times, obtained as the total running time of the queries in the sample divided by the number of queries. In the figures, we report the curves for the different configurations but do not report the standard deviation because in most cases it is smaller than the size of the marker used for distinguishing the different lines. Also, since the main objective is the comparison with the query execution time of the baseline rather than the absolute times, the scale varies for the different experiments. Note that for the baseline configuration, global execution time corresponds to server-side execution time with just the addition of network latency, as it is expected, since in the baseline requires no post processing the client.

Figures 12 and 13 show the global and the server execution time for single and multi-attribute queries, varying the value of k.

Global execution time shows a different trend for point queries over `WAGP` with respect to the one observed for point queries over `OCCP`: at the increase of k, the global execution time decreases for queries over `WAGP` while it increases for queries over `OCCP` (this latter is the trend observed also for queries over `ST` and `AGEP`). The different behavior depends on the interplay between a number of factors: the increase in k leads to a greater number of spurious tuples, but it also leads to queries that access a smaller number of groups, which being of larger size may also benefit more from data compression; for attribute `WAGP`, which is the one with the greatest cardinality, the partitioning leads to a greater probability of having tuples with similar values in the same groups (testified by the low data overhead for range queries reported in Figure 11(b)) and then an improved performance as k grows.

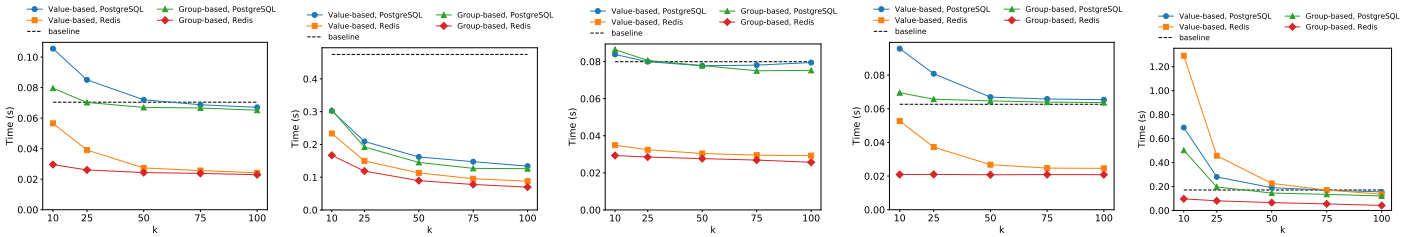
Server execution time is mostly well below the server execution time observed for the baseline and decreases at the increase of k. This is explained by the simpler structure of the queries, which for larger k values provide a smaller number of index values or group ids to extract; the data overhead is greater, but the reduced complexity of the query leads to better server performance. Also, the Redis implementation consistently enjoys lower execution time at the server. This derives from the greater efficiency of Redis in the management of simple data structures. We also note that configurations with group-based indexing are consistently faster than the ones using the value-based indexing. The difference in speed between group-based and value-based indexing decreases when k increases, as larger k implies that a smaller number of index values is generated. As a final observation, we note that point queries exhibit lower query processing times compared to range queries. This is justified by the larger size of the query results of range queries.

Multi-attribute point queries with group-based indexes show a significant improvement in the global execution time. This happens because multi-attribute group-based point queries are more selective than the single-attribute ones, and then require less data transfer, thus producing a saving in the global execution time. We also note that, for multi-attribute point queries for the group-based indexing, the server-side execution time dominates the global execution time, meaning that the time required for data processing at the client is negligible compared to the server-side execution time. Also, with the group-based indexing the conjunction between the conditions on attributes `OCCP`



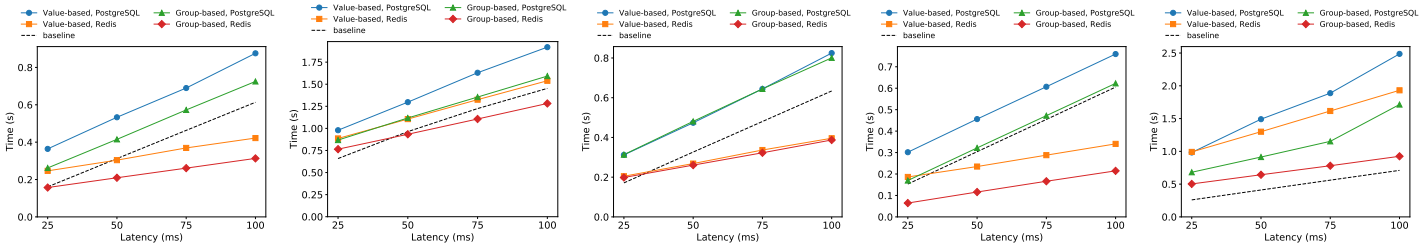
(a) Point queries (WAGP) (b) Range queries (WAGP) (c) Point queries (OCCP) (d) Point queries (WAGP,OCCP) (e) Range queries (WAGP,OCCP)

Fig. 12: Global execution time for single (a,b,c) and multi-attribute (d,e) queries



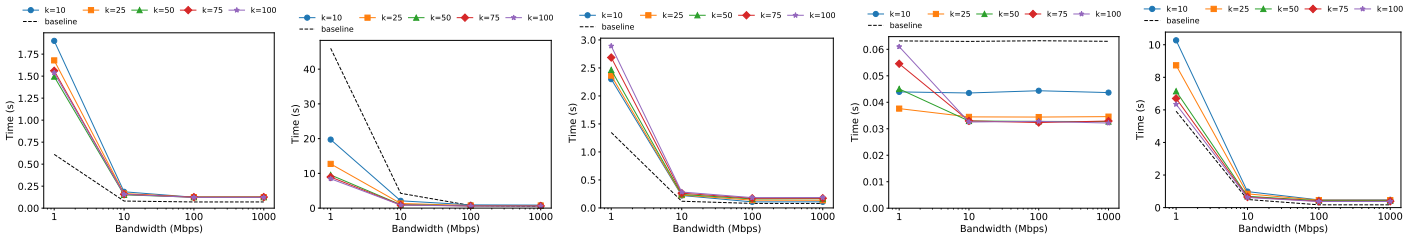
(a) Point queries (WAGP) (b) Range queries (WAGP) (c) Point queries (OCCP) (d) Point queries (WAGP,OCCP) (e) Range queries (WAGP,OCCP)

Fig. 13: Server execution time for single (a,b,c) and multi-attribute (d,e) queries



(a) Point queries (WAGP) (b) Range queries (WAGP) (c) Point queries (OCCP) (d) Point queries (WAGP,OCCP) (e) Range queries (WAGP,OCCP)

Fig. 14: Global execution time for single (a,b,c) and multi-attribute (d,e) queries, varying latency



(a) Point queries (WAGP) (b) Range queries (WAGP) (c) Point queries (OCCP) (d) Point queries (WAGP,OCCP) (e) Range queries (WAGP,OCCP)

Fig. 15: Global execution time for single (a,b,c) and multi-attribute (d,e) queries, varying bandwidth

and WAGP is performed at the client and this produces a saving, since less index values are generated and communicated to the server. This is visible by comparing the global execution time for value-based indexing and for group-based indexing.

With respect to the multi-attribute range queries, we observe similar performance between the baseline and the configurations using group-based indexing (Figure 12(e)), while there is a degradation of performance when using value-based indexing. From the comparison between Figure 12(e) and Figure 13(e) we can see that, for all configurations, a non-negligible amount of time is spent to delete spurious tuples and to transfer the data (we will evaluate bandwidth impact on performance in the following).

Impact of latency. Since latency has a direct impact on performance and usability, we have quantitatively measured its impact repeating the set of experiments illustrated above varying the latency. Figure 14 shows the global execution time of single-attribute and multi-attribute queries, considering k=50 and latency values of 25, 50, 75, and 100 ms, corresponding to round-trip-times equal to twice the latency.[1] These latency values have been selected to mimic a variety of configurations, with the server located in the same geographic region or farther from the client (the range 25-100 ms covers most of the scenarios where a client accesses the servers of a cloud provider).

1. The delay is set using `tc` [21], a utility bundled with `iproute2` [22] that permits to control the Kernel packet scheduler.

The trend for single-attribute and multi-attribute queries are similar. For all five configurations, the execution time grows linearly with the increase of latency, with Redis configurations enjoying a lower slope with respect to PostgreSQL ones (including the baseline, which has been implemented using PostgreSQL). There are two main observations from the experiments, both supporting the applicability of our approach. First, with the execution time increasing linearly at the increase of latency, the net effect is a proportional reduction of the overhead due to grouping. Second, the Redis implementation, enjoying a lighter communication and access protocol, is less affected by latency increase.

Impact of bandwidth. We ran a dedicated set of experiments varying the bandwidth between client and server, to evaluate its effect on query execution time and hence the applicability of the approach in low bandwidth scenarios. We have then repeated the set of experiments discussed above considering bandwidth values of 1 Mbps, 10 Mbps, 100 Mbps, and 1 Gbps (this latter being the one considered before). Figure 15 reports the global execution time for single-attributes and multi-attribute queries for different values of k, varying the bandwidth. Queries are issued using our group-based implementation for Redis. As it is visible from the figures, the overhead is negligible for bandwidth of at least 10 Mbps (which can be assumed to cover the wide majority of configurations, given that evolution of network technology is making available, in most scenarios, communication channels with bandwidth above 100 Mbps). It is to note also that for range and multi-attribute queries, the data compression and serialization provided by our implementation and the greater efficiency of Redis, in scenarios with low bandwidth, produce improvements in the overall execution time with respect to the one offered by the baseline PostgreSQL plaintext implementation.

6 Related work

The problem of supporting query evaluation over encrypted data stored off-premises has been widely studied. Existing approaches that address this problem rely on the use of specific cryptographic primitives or on the definition of indexes.

The cryptographic primitives supporting searches over encrypted data include property-preserving encryption (e.g., [23], [24], [25], [26]), searchable symmetric encryption (SSE) and range SSE (e.g., [27], [28], [29]), and fully homomorphic encryption (e.g., [30], [31], [32]). These approaches provide a different trade-off among efficiency, security, and the kind of supported queries (e.g., [33], [34]). In particular, maintaining plaintext functionality (e.g., preserving order of values) in the encryption makes the encryption vulnerable to inferences, as the information carried by such functionality is leaked. Also, while considerable progress has been made in the field, the computational overhead of stronger cryptographic primitives still results too cumbersome for most database applications. Our proposal differs from these approaches mainly because we use an auxiliary indexing structure on the client, and store together in a single block groups of tuples of uniform cardinality and size. These aspects mitigate the possible leakages deriving from the execution of cryptographic functions [35] and from the retrieval of single tuples.

Indexes are metadata defined over attributes frequently involved in query evaluation (e.g., [4], [5], [36], [37], [38]). Indexes are stored together with the encrypted data and can be used to efficiently retrieve the data to be returned in response to a query. Different approaches to indexing have been investigated. Some indexing techniques are built over a single attribute of the outsourced relation (e.g., [5]), and therefore can only support queries defined on such an attribute. Other solutions support indexes on multiple attributes. These include the work in [39] that, similarly to our proposal, provides indexes at the group level, but with the aim to find a balance between the number of spurious tuples and the protection given by the entropy in the query results; hence, it considers a different setting of the problem. Other approaches rely on tree-based structures (e.g., R-trees and KD-trees) that have been designed to efficiently support queries over plaintext data (e.g., nearest neighbor searches in spatial applications [40]). The problem would then be how to efficiently and securely traverse a tree-based structure whose nodes are encrypted with the same cryptographic primitives used for protecting data. Different indexing solutions focus on different aspects of this problem, such as the definition of new search algorithms, the definition of novel cryptographic techniques that support the tree-traversal procedure of, for example, R-trees, or the definition of novel tree-based structures supporting range queries (e.g., [41], [42], [43], [44]). These proposals mainly focus on the efficiency aspect or on the cryptographic techniques and do not address the problem of protecting against frequency-based attacks, do not support flat grouping of the data to be protected, and do not consider the storage of data in a key-value database.

Other lines of work in the context of ensuring some form of confidentiality in data outsourcing includes: the use of trusted hardware for protecting query exection [2], [3]; the fragmentation of data for their external storage (so to avoid or limit encryption when what is to be protected is the association among the data rather than their values) [45], [46], [47]; and the protection of the confidentiality of the accesses and their patterns, with different variations of ORAM-based solutions (e.g., Path-ORAM) typically relying on data re-allocation to break the otherwise fixed correspondence between data and their physical storage location (which comes at the price of significant overhead and limitations in query execution). While sharing the scenario of data outsourcing to not fully trusted services, these approaches address therefore a different problem.

7 Conclusions

We have presented an approach for the definition of multi-attribute indexes that enables the execution of point and range queries over encrypted data outsourced to an external cloud provider. The proposed approach to index construction provides both flattening and collisions on any combination of index values, and ensures an effective execution of queries. Our experimental evaluation considers the storage of data on both an external relational database (PostgreSQL)

and a key-value database (Redis), and shows the effectiveness of the proposal, thus supporting its application in real-world scenarios.

ACKNOWLEDGEMENTS

This work was supported in part by the EC under projects Chips JU EdgeAI (101097300) and GLACIATION (101070141), by the Italian MUR under PRIN project POLAR (2022LA8XBH), and by project SERICS (PE00000014) under the MUR NRRP funded by the EU - NextGenerationEU.

REFERENCES

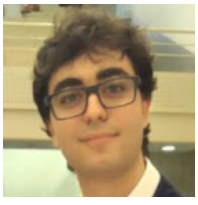
[1] G. Poh, J. Chin, W. Yau, K. Choo, and M. Mohamad, "Searchable symmetric encryption: Designs and challenges," *ACM CSUR*, vol. 50, no. 3, pp. 1–37, 2017.
[2] A. Arasu, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, and R. Ramamurthy, "Transaction processing on confidential data using cipherbase," in *Proc. of ICDE*, 2015, pp. 435–446.
[3] S. Bajaj and R. Sion, "TrustedDB: A trusted hardware-based database with privacy and data confidentiality," *IEEE TKDE*, vol. 26, no. 3, pp. 752–765, 2014.
[4] E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Balancing confidentiality and efficiency in untrusted relational DBMSs," in *Proc. of ACM CCS*, 2003, pp. 93–102.
[5] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra, "Executing SQL over encrypted data in the database-service-provider model," in *Proc. of ACM SIGMOD*, 2002, pp. 216–227.
[6] K. LeFevre, D. DeWitt, and R. Ramakrishnan, "Mondrian multidimensional *k*-anonymity," in *Proc. of ICDE*, 2006, pp. 25–36.
[7] A. C. Yao, "New algorithms for bin packing," *J. ACM*, vol. 27, no. 2, pp. 207–227, 1980.
[8] D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O'Hara, F. Saint-Jacques, and G. S. Y. Kai, "Roaring bitmaps: Implementation of an optimized software library," *Softw. Pract. Exp.*, vol. 48, no. 4, pp. 867–895, 2018.
[9] The pandas development team, "Pandas-dev/pandas: Pandas," https://doi.org/10.5281/zenodo.3509134, 2020.
[10] The Apache Software Foundation, "Apache Arrow," https://arrow.apache.org/, 2021.
[11] G. Van Rossum, "The python library reference, release 3.10.4," https://docs.python.org/3/library/pickle.html, 2022.
[12] J. Seward, "bzip2 and libbzip2," http://www.bzip.org, 1996.
[13] Facebook, "Zstandard," https://facebook.github.io/zstd/, 2021.
[14] Docker inc., "Docker-compose," https://docs.docker.com/compose/, 2021.
[15] M. Bayer, "SQLAlchemy," in *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*, A. Brown and G. Wilson, Eds., 2012.
[16] Redis Inc., "redis-py," https://redis.readthedocs.io/en/latest/, 2021.
[17] U.S. Bureau of the Census, "Public use microdata sample. individual dataset of all us. 1-year version of acs 2019," https://www2.census.gov/programs-surveys/acs/data/pums/2019/1-Year, 2019.
[18] ——, "Public use microdata sample. individual dataset of all us. 1-year version of acs 2018," https://www2.census.gov/programs-surveys/acs/data/pums/2018/1-Year, 2019.
[19] Kaggle, "Acquire valued shoppers challenge, transactions dataset," https://www.kaggle.com/c/acquire-valued-shoppers-challenge/data?select=transactions.csv.gz, 2014.
[20] E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Balancing confidentiality and efficiency in untrusted relational DBMSs," in *Proc. of ACM CCS*, 2003.
[21] "tc(8) – Linux manual page," https://man7.org/linux/man-pages/man8/tc.8.html, 2022.
[22] "iproute2 – Ubuntu man pages," https://launchpad.net/ubuntu/focal/+package/iproute2, 2021.
[23] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, "Order-preserving symmetric encryption," in *Proc. of EUROCRYPT*, 2009, pp. 224–241.
[24] R. .Popa, F. H. Li, and N. Zeldovich, "An ideal-security protocol for order-preserving encoding," in *Proc. of IEEE S&P*, 2013, pp. 463–477.
[25] D. Li, S. Lv, Y. Huang, Y. Liu, T. Li, Z. Liu, and L. Guo, "Frequency-hiding order-preserving encryption with small client storage," *PVLDB*, vol. 14, no. 14, pp. 3295–3307, 2021.
[26] A. Roy Chowdhury and P. Ramanathan, "Public order preserving cipher generation scheme for distributed computing," in *Proc. of CCS*, 2018, pp. 2273–2275.
[27] G. Poh, J. Chin, W. Yau, K. Choo, and M. Mohamad, "Searchable symmetric encryption: Designs and challenges," *ACM CSUR*, vol. 50, no. 3, pp. 1–37, 2017.
[28] C. Bösch, P. Hartel, W. Jonker, and A. Peter, "A survey of provably secure searchable encryption," *ACM CSUR*, vol. 47, no. 2, pp. 1–51, 2015.
[29] F. Falzon, E. Markatou, Z. Espiritu, and R. Tamassia, "Attacks on encrypted range search schemes in multiple dimensions," Cryptology ePrint Archive, 2022, https://eprint.iacr.org/2022/090.pdf.
[30] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. of ACM STOC*, 2009, pp. 169–178.
[31] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," in *Proc. of EUROCRYPT*, 2011, pp. 129–148.
[32] Microsoft, "Microsoft SEAL," 2021, https://www.microsoft.com/en-us/research/project/microsoft-seal.
[33] R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting confidentiality with encrypted query processing," in *Proc. of SOSP*, October 2011, pp. 85–100.
[34] M. Naveed, S. Kamara, and C. Wright, "Inference attacks on property-preserving encrypted database," in *Proc. of ACM CCS*, 2015, pp. 644–655.
[35] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, M. Garofalakis, and C. Papamanthou, "Practical private range search in depth," *ACM TODS*, vol. 43, no. 1, pp. 1–52, 2018.
[36] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis, "Practical private range search revisited," in *Proc. of ACM SIGMOD*, 2016, pp. 185–198.
[37] H. Van Tran, T. Allard, L. d'Orazio, and A. El Abbadi, "FRESQUE: A scalable ingestion framework for secure range query processing on clouds," in *Proc. of EDBT*, 2021, pp. 205–216.
[38] B. Hore, S. Mehrotra, and G. Tsudik, "A privacy-preserving index for range queries," in *Proc. of VLDB*, 2004, pp. 720–731.
[39] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu, "Secure multidimensional range queries over outsourced data," *The VLDB Journal*, vol. 21, no. 3, pp. 333–358, 2012.
[40] J. Wang, S. Wu, H. Gao, J. Li, and B. Ooi, "Indexing multi-dimensional data in a cloud system," in *Proc. of SIGMOD*, IN, USA, June 2010.
[41] B. Wang, Y. Hou, and M. Li, "QuickN: Practical and secure nearest neighbor search on encrypted large-scale data," *IEEE TCC*, vol. 10, no. 3, pp. 2066 – 2078, 2022.
[42] B. Wang, Y. Hou, M. Li, H. Wang, and H. Li, "Maple: Scalable multi-dimensional range search over encrypted cloud data with tree-based index," in *Proc. of ACM ASIACCS*, 2014, pp. 111–122.
[43] Z. Wu and K. Li, "VBTree: Forward secure conjunctive queries over encrypted data for cloud computing," *The VLDB Journal*, vol. 28, no. 1, pp. 25–46, 2019.
[44] S. Wu, Q. Li, G. Li, D. Yuan, X. Yuan, and C. Wang, "ServeDB: Secure, verifiable, and efficient range queries on outsourced database," in *Proc. of ICDE*, 2019, pp. 626–637.
[45] G. Cormode, D. Srivastava, T. Yu, and Q. Zhang, "Anonymizing bipartite graph data using safe groupings," *PVLDB*, vol. 1, no. 1, pp. 833–844, 2008.
[46] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Fragments and loose associations: Respecting privacy in data publishing," *PVLDB*, vol. 3, no. 1, pp. 1370–1381, 2010.
[47] X. Xiao and Y. Tao, "Anatomy: Simple and effective privacy preservation," in *Proc. of VLDB*, 2006, pp. 139–150.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2024.3408905

Sabrina De Capitani di Vimercati is a professor at the Università degli Studi di Milano, Italy. Her research interests are in data security and privacy. She has published more than 230 papers in journals, conference proceedings, and books. She has been a visiting researcher at SRI International, CA, USA, and George Mason University, VA, USA.
https://decapitani.di.unimi.it

Dario Facchinetti is a post-doctoral researcher at the Università degli Studi di Bergamo, Italy. His work ranges from the integration of security features in mobile, database and cloud systems, to policy and privacy management. He is interested in access control and sandboxing techniques.

Sara Foresti is a professor at the Università degli Studi di Milano, Italy. Her research interests are in data security and privacy. She has published more than 100 papers in journals, conference proceedings, and books. She has been a visiting researcher at George Mason University, VA, USA. She chairs the IFIP WG 11.3 on Data and Applications Security and Privacy.
https://foresti.di.unimi.it



Gianluca Oldani is currently pursuing the Ph.D. degree with the Università degli Studi di Bergamo, Italy. His research interests include web security, distributed technologies, and data privacy.

Stefano Paraboschi is a professor at the Università degli Studi di Bergamo, Italy. His research focuses on information security and privacy, Web technology for data intensive applications, XML, information systems, and database technology. He has been a visiting researcher at Stanford University and IBM Almaden, CA, USA, and George Mason University, VA, USA.
https://cs.unibg.it/parabosc

Matthew Rossi is currently pursuing the Ph.D. degree with the Università degli Studi di Bergamo, Italy. From 2019 to 2020, he was a Research Assistant with the Department of Information Engineering, Università degli Studi di Bergamo. His research interest includes the integration of security features in mobile systems, policy management and privacy of outsourced data.

Pierangela Samarati is a professor at the Università degli Studi di Milano, Italy. Her main research interests are in data protection, security, and privacy. She has published more than 290 papers in journals, conference proceedings, and books. She has been a visiting researcher at Stanford University, CA, USA, SRI International, CA, USA, and George Mason University, VA, USA. She is a Fellow of ACM, IEEE, and IFIP.
https://samarati.di.unimi.it